An Interval-based Abstraction for Quantifying Information Flow

Technical Report: TR-10-04

Chunyan Mu 1  David Clark 2

Department of Computer Science
King’s College London
The Strand, London, UK

Abstract

In a batch program, information about confidential inputs may flow to insecure outputs. The size of this leakage, considered as a Shannon measure, may be automatically and exactly calculated via probabilistic semantics as we have shown in our earlier work. This approach works well for small programs with small state spaces. As the scale increases the calculation suffers from a form of state space explosion and the time complexity grows. In this paper we scale up the programs and state spaces that can be handled albeit at the cost of replacing an exact result with an upper bound. To do this we introduce abstraction on the state space via interval-based partitions, adapting an abstract interpretation framework introduced by Monniaux. The user can define the partitions and the more coarse the partitions, the more coarse the resulting upper bound. In this paper we summarise our previous contribution, define the abstract interpretation, show its soundness, and prove that the result of an abstract computation is always an upper bound on the true leakage, i.e. is a safe estimate. Finally we illustrate the approach by means of some examples.

Keywords: Security, Abstraction, Information Flow, Measurement, Language

1 Introduction

Information-flow security enforces limits on the use of information propagation in programs. The goal of information flow security is to ensure that the information propagates throughout the execution environment without security violations so that no secure information is leaked to public outputs. The traditional theory based reasoning and analysis of software systems are not concerned with security measurement. Quantitative information flow (QIF) proposes to determine how much

1 Email: chunyan.mu@kcl.ac.uk
2 Email: david.j.clark@kcl.ac.uk
information flows from confidential inputs to public outputs. It is concerned with measuring the amount of information flow caused by interference between variables by using information theoretic quantities. Quantitative analysis therefore relaxes the well known non-interference [17] property by introducing a new policy: the program is secure if the amount of information flow from confidential inputs to public outputs is not too big.

In [29], we presented an automatic analyzer for measuring information flow within software systems. We considered the mutual information between a high security variable at the beginning of a batch program and a low one at the end of the program, conditioned on the initial values of low, as the measure of how much of the initial secret information is leaked by executing the program [4]. We incorporated the leakage computation into Kozen’s probabilistic semantics to build an automatic analysis tool. This provided a more precise analysis. It took as input a probability distribution on the initial store and calculated a probability distribution on the final store when the program sometimes terminates. In fact it calculates a probability distribution at each program point. These can then be used to calculate the exact leakage. Specifically, while-loops were handled by applying the more general definition of entropy of generalized distributions [30] and related properties in order to provide a more precise analysis when incorporating elapsed observed time into the analysis. The drawback of this approach is that it is lacking in abstraction techniques, and time complexity can become large in certain circumstances. One of the time complexity issues is from mutual information computation [29]. Another one is introduced by applying concrete semantics to while loop, which can be improved by applying the abstraction considered in this paper.

We introduce here an approximation method based on measurable partitions and Monniaux’s abstract probabilistic semantics [28]. We define a basic abstract domain using interval-based partitions with weights to abstract the measure space. We then define abstract semantic functions which transform the abstract state space in the abstract domain. The definition of leakage upper bound due to such semantics is presented. We introduce uniformalization to provide upper bounds on the leakage computation in the framework of information theory. The security property we consider is an a priori bound on leakage, that is, the program is regarded as sufficiently secure if the leakage is less than the a priori bound. Since the actual leakage in the program is required to be less than or equal to the a priori bound, the analysis is safe if we find an upper bound on the leakage [4].

The rest of the paper is organized as follows. Section 2 explains the concrete semantics and leakage computation used in our earlier work. In Section 3, we present
a method to approximate our concrete leakage analysis and prove some properties of this. Finally, we present related work and draw conclusions in Section 5 and 6.

2 Leakage Analysis via Probabilistic Semantics

We have developed an automatic leakage analysis system in [29] without abstraction. This previous work is summarized here for the sake of clarity.

2.1 Measure space and programs

Assume that the tuple of program variables $\vec{X}$ range over the state space $\Omega$. The denotational semantics of a command is a mapping from the set $X$ of possible environments before a command into the set $X'$ of possible environments after the command. These spaces updated by semantic transformation functions, can be used to calculate leakage at each program point. Some useful definitions from [33] are reviewed as follows. A measure space is a triple $(\Omega, \mathcal{B}, \mu)$, where $\Omega$ is a set, $\mathcal{B}$ is a $\sigma$-algebra of subsets of $\Omega$, $\mu$ is a nonnegative, countably finite additive set function on $\mathcal{B}$. A $\sigma$-algebra is a set of subsets of a set $X$ that contains $\emptyset$, and is stable under countable union and complementation. A set $X$ with a $\sigma$-algebra $\sigma_X$ defined on it is called a measurable space and the elements of the $\sigma$-algebra are the measurable subsets. If $X$ and $X'$ are measurable spaces, $f : X \rightarrow X'$ is a measurable function if for all $W$ measurable in $X'$, $f^{-1}(W)$ is measurable in $X$. A positive measure is a function $\mu$ defined on a $\sigma$-algebra $\sigma_X$, which is countable additive, i.e. if taking $(E_n)_{n \in \mathbb{N}}$ a disjoint collection of elements of $\sigma_X$, then $\mu(\bigcup_{n=0}^{\infty} E_n) = \sum_{n=0}^{\infty} \mu(E_n)$. A probability measure is a positive measure of total weight 1. A sub-probability measure has total weight less than or equal to 1.

2.2 The language

The language we considered is standard, presented in Table 1.

<table>
<thead>
<tr>
<th>$c \in \text{Cmd}$</th>
<th>$x \in \text{Var}$</th>
<th>$e \in \text{Exp}$</th>
<th>$b \in \text{BExp}$</th>
<th>$n \in \text{Num}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c ::= \text{skip}$</td>
<td>$x ::= e$</td>
<td>$</td>
<td>c_1 ; c_2$</td>
<td>$</td>
</tr>
<tr>
<td>$c ::= x[n]e_1 + e_2</td>
<td>e_1 - e_2</td>
<td>e_1 * e_2</td>
<td>e_1/e_2$</td>
<td></td>
</tr>
<tr>
<td>$b ::= \neg b$</td>
<td>$e_1 &lt; e_2</td>
<td>e_1 \leq e_2</td>
<td>e_1 = e_2$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 The language
2.3 Probabilistic semantics and leakage computation

In [29], we used Kozen’s probabilistic semantics for a while language [20] to calculate the probability distribution on the final state from knowledge of the initial state. We then used this to calculate the quantity of leakage from any initial variable to any final variable.

Information theory introduced the definition of entropy, $H$, to measure the average uncertainty in random variables. Shannon’s measures were based on a logarithmic measure of the unexpectedness of a probabilistic event (random variable). The unexpectedness of an event which occurred with some non-zero probability $p$ was $\log_2 \frac{1}{p}$. Hence the total information carried by a set of events was computed as the weighted sum of their unexpectedness: $H = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$. Assume we have two types of input variables: $H$ (confidential) and $L$ (public), and the inputs are equipped with probability distributions, so the inputs can be viewed as a joint random variable $(H, L)$. The semantic function for programs maps the state of inputs to the state of outputs. We present the basic leakage definition due to Clark, Hunt and Malacaria [4] for programs as follows.

**Definition 2.1 (Leakage)** Let $H$ be a random variable in high security inputs, $L$ be one in low security inputs, and let $L'$ be a random variable in the output observation, the secure information flow (or interference) is defined by $I(L'; H|L)$, i.e. the conditional mutual information between the output and the high input given knowledge of the low output. Note that for deterministic programs, we have $I(L'; H|L) = H(L'|L)$, i.e. interference between the uncertainty in the public output given knowledge of the low input.

The denotational semantics for measure space transformations are in the following forms:

$$Val \triangleq (\Omega, B, \mu) \quad \Sigma \triangleq \vec{X} \rightarrow Val$$

$$C[\cdot] : Cmd \rightarrow (\Sigma \rightarrow \Sigma) \quad E[\cdot] : Exp \rightarrow (\Sigma \rightarrow Val)$$

$$B[\cdot] : BExp \rightarrow (\Sigma \rightarrow \Sigma)$$

Consider the program as a state-transformation machine, and assume that the tuple of program variables $\vec{X}$ range over the state space $\Omega$. The program variables $\vec{X}$ satisfy some joint distribution $\mu$ on program inputs: $\mu(\vec{X})$ assigned to each $\vec{X} \in \Omega; \mu(\vec{X}) \geq 0; \sum_{\vec{X} \in \Omega} \mu(\vec{X}) = 1$. Let $[\cdot]$ denote any measure space transformer given by a semantic function, precisely, $[\cdot]$ means $[\cdot] : CB \in \text{Cmd} \cup \text{BExp}$, i.e. $[\cdot] : \Sigma \rightarrow \Sigma'$. In Kozen’s probabilistic semantics, a program transformation function
Mu and Clark

maps distributions $\mu$ on $\vec{X}$ to distributions $\mu' = [\cdot](\mu)$ on $\vec{X}$. Let $X$ range over $\Omega$ in $\Sigma$, $X'$ range over $\Omega$ in $\Sigma'$, $\mu$ is in $\Sigma$, and $\mu'$ is in $\Sigma'$, i.e. $X$ and $X'$ denote the measure space over $\vec{X}$ before and after $[\cdot]$. Consider the general form of an action of one elementary operator:

- $X \rightarrow (X' = [\cdot](X))$
- $\mu \rightarrow \mu'$
- $\mu'(X') = \sum_{X \in \Omega} \mu(X) \delta_{X', [\cdot](X)}$ where,
  
  \[
  \begin{cases}
  \delta_{a,b} = 1 & \text{iff } a = b \\
  \delta_{a,b} = 0 & \text{iff } a \neq b
  \end{cases}
  \]

We concentrate on the distributions and present the space transformation functions offered by semantic functions by using Lambda Calculus and the notation of inverse function following [28] in Table 2.

\[
[x := e](\mu) \triangleq \lambda W.\mu([x := e]^{-1}(W))
\]
\[
[c_1; c_2](\mu) \triangleq [c_2] \circ [c_1](\mu)
\]
\[
[\text{if } b \text{ then } c_1 \text{ else } c_2](\mu) \triangleq [c_1] \circ [b](\mu) + [c_2] \circ [\neg b](\mu)
\]
\[
[\text{while } b \text{ do } c](\mu) \triangleq [\neg b](\lim_{n \to \infty}(\lambda \mu'.\mu + [c] \circ [b](\mu'))^n)(\lambda X.\bot)
\]

where, $[B](\mu) = \lambda X.\mu(X \cap B)$

Table 2

Probabilistic Denotational Semantics of Programs

Due to the measurability of the semantic functions, for all measurable $W \in X'$, $[x := e](W)$ is measurable in $X$. The function $[B]$ for boolean test $B$ defines the set of environments matched by the condition $B$: $[B](\mu) = \lambda X.\mu(X \cap B)$, which causes the space to split apart, $X \cap B$ denotes the part of the space $X$ which makes boolean test $B$ to be true. Conditional statement is executed on the conditional probability distributions for either the true branch or false branch: $[c_1] \circ [b](\mu) + [c_2] \circ [\neg b](\mu)$. In the while loop, the measure space with distribution $\mu$ goes around the loop, and at each iteration, the part that makes test $b$ false breaks off and exits the loop, while the rest of the space goes around again. The output distribution $[\text{while } b \text{ do } c](\mu)$ is thus the sum of all the partitions that finally find their way out. Note that these partitions are part of the space when the loop partially terminates, which implies the outputs are partially observable and hence produce an incomplete distribution. The incomplete distribution incorporates the non-termination part (the rest of the space) to be a complete distribution finally. For the case that the loop is always
non-terminating, ⊥ is returned and leakage is 0.

In what follows we present some of concrete semantic operations with leakage analysis.

- **Assignment** Assignment updates the state such that the measure space of assigned variable $x$ is updated to become the measure space for expression $e$. For example, if there is no low input, the secure information leaked to low security variable $x$ after command $[x := e]$ is given by $L_{[x := e]} = H(\mu_e)$. The secure information contained in $e$ is considered as the entropy of its distribution $\mu_e$.

- **Sequential composition command** The distribution transformation function for the sequential composition operator is obtained via functional composition:

$$[c_1; c_2](\mu) = [c_2] \circ [c_1](\mu).$$

- **Conditionals** Conditional tests cause a partition of the measure space into values that make the boolean $b$ true and those that make it false: let $[b]_s$ denote the standard state transformer of boolean test, $P_0 \subseteq X$, $\forall x \in P_0, [b]_s x = \text{tt}$, i.e. $k$ makes test $b$ be true; $P_1 \subseteq X$, $\forall x \in P_1, [b]_s x = \text{ff}$, i.e. $k$ makes test $b$ be false, and $P_0 \cup P_1 = X \land P_0 \cap P_1 = \emptyset$. Let $\mu_b(\text{tt}) = \sum_{x \in P_0} \mu(x)$ denote the probability that $b$ is true, and $\mu_b(\text{ff}) = \sum_{x \in P_1} \mu(x)$ denote the probability that $b$ is false under the current space. Let $P_0 = \{p_0 = \mu_b(\text{tt})\}$, $P_1 = \{p_1 = \mu_b(\text{ff})\}$ denote the partitions due to the test $b$, and $Q_l^0 = \{q_{00}, \ldots, q_{0m}\}$, $Q_l^1 = \{q_{10}, \ldots, q_{1n}\}$ denote the normalized distribution of the low security variable $l$ inside the partitions.

The semantic function is given by $[\text{if } b \text{ then } c_1 \text{ else } c_2](\mu) = [c_1] \circ [b](\mu) + [c_2] \circ [\neg b](\mu)$. The leakage into $l$ due to the if statement can be computed by:

$$L_{[\text{if }]} = \tilde{H}(P_0 \cup P_1) + \tilde{H}(Q_l^0 \cup Q_l^1 \mid P_0 \cup P_1)$$

where $\tilde{H}$ denotes the entropy of generalised distributions [29].

**Example 2.2** To show how the method works, let us consider a simple program [29].

```plaintext
if (h==0) then l=0 else l=1;
```

Assume $h$ is a 32-bit high security variable with possible values $0, \ldots, s^k - 1$ under uniform distribution, $l$ is a low security variable. The boolean test splits the original space and we get $P_0 = \{\frac{1}{2^{32}}\}$, $P_1 = \{1 - \frac{1}{2^{32}}\}$, and $Q_l^0 = \{\mu_l(0)\} = \{\frac{1}{2^{32}}\}$, $Q_l^1 = \{\mu_l(1)\} = \{1 - \frac{1}{2^{32}}\}$. The resulting set of distribution transformation is obtained as:
The distribution of low security variable \( l \) after the execution of the program is as follows: 
\[
\begin{pmatrix}
0 & 1/2^{32} \\
1 & 1 - 1/2^{32}
\end{pmatrix}
\]
and the information leakage due to this example can be computed by:
\[
H_{[\{1\}]} = \tilde{H}(P_0 \cup P_1) + \tilde{H}(Q_0 \cup Q_1 | P_0 \cup P_1)
\]
\[
= \tilde{H}(\{1/2^{32}\} \cup \{1 - 1/2^{32}\}) + 0 = 7.8 \times 10^{-9}
\]

It is clear that this example just releases a little bit information to the public.

The leakage analysis result also agrees with our intuition: the possibility of \( h = 0 \) is quite low and the uncertainty of \( h \) under condition \( h \neq 0 \) is still big, i.e. only small information released to the public.

- **While Loop** In the while loop, the distribution goes around the loop, at each round, the part that makes test \( b \) false breaks off and exits the loop, and the rest goes around again. Command \([\text{while } b \text{ do } c]\) can be rewritten as \([\text{if } \neg b \text{ then skip else } \{c; \text{while } b \text{ do } c\}]\). The operator for loops can be described in a recursive way, we have:

\[
\begin{aligned}
[\text{while } b \text{ do } c]\,^0(\mu) &= \mu \\
[\text{while } b \text{ do } c]\,^n(\mu) &= [\neg b](\mu) + [b] \circ [b]([\text{while } b \text{ do } c]\,^{n-1}(\mu))
\end{aligned}
\]

For the always terminating loops, the output distribution is the sum of all the sub-distributions that fail the conditional test on each iteration and find their way out of the loop until \([b](\mu) = \emptyset\). The loop is considered as partially/completely non-terminated when no new partitions are produced but the test is still satisfied with respect to the partial/whole space. Consider a terminating loop as a sub-measure transformer which builds a set of accumulated incomplete distributions, i.e. due to the \( k^{th} \) iteration, \( \mathcal{P}([\text{while } b \text{ do } c]) = \bigcup_{0 \leq i \leq k} \mathcal{P}_i([\text{while } b \text{ do } c]) \), where \( k \leq n \), and \( n \) is the maximum number of the partitions produced by the terminating loops. Let
The leakage computation of the loop given by addition of the entropy of the union of the boolean test for each iteration and the sum of the entropy of the loop body for each weighted sub-probability measures by applying the definition of entropy of partitions [32, 30]:

\[ L_{\text{while}}(k) = \tilde{H}(P_0 \cup \ldots \cup P_k) + \tilde{H}(Q_0^L \cup \ldots \cup Q_k^L | P_0 \cup \ldots \cup P_k) \]

where \( P_i \) is the event that the loop test \( b \) is true until the \( i \)th iteration, \( b^i \) denotes the value of the boolean test \( b \) at the \( i \)th iteration. Consider the union of the decompositions

\[ P = (P_0 \cup P_1 \cup \ldots \cup P_k)_{0 \leq k \leq n} = (\{p_0\} \cup \{p_1\} \cup \ldots \cup \{p_k\})_{0 \leq k \leq n} \]

the events \( P_0, \ldots, P_k \) build the out-going partitions of the states for a while loop.

The leakage computation of the loop given by addition of the entropy of the union of the boolean test for each iteration and the sum of the entropy of the loop body for each weighted sub-probability measures by applying the definition of entropy of partitions [32, 30]:

\[ L_{\text{while}}(k) = \tilde{H}(P_0 \cup \ldots \cup P_k) + \tilde{H}(Q_0^L \cup \ldots \cup Q_k^L | P_0 \cup \ldots \cup P_k) \]

where \( P_i = \{p_i\} \), thus \( \tilde{H}(P_i) = \log_2 \frac{1}{p_i} \). The leakage computation formula works for terminating, partially terminating, and non-terminating loops. Further details and examples refer to [29].

3 Abstraction on Information Flow Measurement

To address the problem of tractability when the size and number of variables get large, we present a method to help us approximate our concrete analysis for leakage. Firstly, we define an abstraction on the measure space. The abstraction technique considered is partitioning of the concrete measure space into blocks. Each block is described by intervals and has a coefficient which is the upper bound on measures. Secondly, abstract semantic operations are applied on these partitions. The abstract transformation behavior of the distributions is described by the abstract transition function \([\cdot] \#\) on the abstract objects based on Monniaux’s [28] abstract probabilistic semantics. To guarantee security, the analysis is safe if we find an upper bound on the leakage. Our analysis is required to over estimate the leakage and never under estimate it. Therefore, we introduce uniformization to estimate the abstract spaces to provide safe bounds on the entropy computation. The leakage computation is obtained by estimating the abstract space to maximise the entropy and thus give safe (i.e. upper) bounds on the entropy.
3.1 Measurable Partitions and Abstract Domain

In the first step, we aim to provide an abstraction on the measure space transformation which provides a basis for the leakage computation. The basic principle of the abstraction here is to collapse sets of concrete states into single abstract states by doing interval-based partitions on the concrete space. We give our basic abstract domain, expressing the above idea formally, and then take Monniaux’s \[28\] framework of abstract probabilistic semantics to construct abstract operations on the abstract domains.

3.2 Concrete Lattice

The concrete space we consider is a finite measure space denoted by \(X(\Omega, \mathcal{B}, \mu)\).

**Definition 3.1** [Concrete lattice \(X\)] The \(\sigma\)-algebra \(\mathcal{B}\) of a finite measure space \(X\) forms a complete lattice in which we define a partial order on \(\mathcal{B}\) as follows:

\[
\forall x_1, x_2 \in \mathcal{B}, x_1 < x_2 \text{ iff } \mathcal{H}(x_1) \leq \mathcal{H}(x_2)
\]

where \(\mathcal{H}\) denotes Shannon entropy. We now define an equivalence relation on \(\mathcal{B}\), we say that

\[
x_1 \simeq x_2 \text{ iff } \mathcal{H}(x_1) = \mathcal{H}(x_2)
\]

Clearly, the lower bound \(l\) on \(\mathcal{B}\) is given as:

\[
l = \{x \mid \mathcal{H}(x) = 0, \ x \in \mathcal{B}\}
\]

and the upper bound \(u\) on \(\mathcal{B}\) is given as:

\[
u = \{x \mid \mathcal{H}(x) = \log_2 N, \ x \in \mathcal{B}\}
\]

where \(N\) is the size of the concrete measure space.

Note that the upper bound \(u\) is obtained when \(x\) is under uniform distribution: \(\mathcal{H}(x) = \log_2 N\).

3.3 Measurable Partitions

Intuitively, our abstract domain is based on partitions of the measure space. Any collection of non-empty disjoint sets that cover a measure space \(X\) is said to be a partition of \(X\).

**Definition 3.2** [Partition of a space c.f. \[32\]] A partition of space \(X\) is defined as a family \(\xi = \{E_i \mid i \in I\}\) of nonempty disjoint subsets of \(X\), whose union is \(X\): \(\bigcup_{E_i \in \xi} E_i = X\). The elements \(E_i\) are called the blocks of \(\xi\).
This can be equivalently considered as a surjection $\phi : X \rightarrow X/\xi$, where each point of the space $X/\xi$ corresponds to a set $E_i$ in $\xi$. Intuitively, $\phi$ is the map taking each point $x \in X$ to the element of $X/\xi$ in which it is contained.

**Definition 3.3** [Measurable partition c.f. [32]] The quotient $\sigma$-algebra on $X/\xi$ is defined as the pushforward under $\phi$ of the $\sigma$-algebra on $X$. A subset $M \subset X/\xi$ is measurable if the inverse image $\phi^{-1}(M) \subset X$ is measurable. Furthermore, $\phi$ is called a measurable partition function, and the partition $\xi$ is called a measurable partition.

Partitions of sets can be viewed as random variables: if $X$ is a finite set on which there is a $\sigma$-algebra, and $\xi = \{E_1, \ldots, E_n\}$ is a partition of $X$, $W(X)$ denotes the weight of $X$, $W(E_i)$ denotes the weight of $E_i$, then $p = (W(E_1)/W(X), \ldots, W(E_n)/W(X))$ is a discrete probability distribution. The Shannon entropy of $p$ equals the Shannon entropy of $\xi$ according to Rokhlin [32].

**Definition 3.4** [Entropy of partition c.f. [32]] The entropy of partition $\xi = \{E_1, \ldots, E_n\}$ on $X$ is defined as

$$ H(\xi) = H\left(\frac{W(E_1)}{W(X)}, \ldots, \frac{W(E_n)}{W(X)}\right) $$

where $W(X)$ denotes the weight of $X$, $W(E_i)$ denotes the weight of $E_i$, and $H$ denotes Shannon entropy.

For example, consider that the measure space $X$ is as follows:

$$ x \mapsto \begin{cases} 0 \rightarrow 0.2 & 1 \rightarrow 0.2 \\ 2 \rightarrow 0.4 & 3 \rightarrow 0.2 \end{cases} $$

Let us take a partition $\xi = \{E_1, E_2\}$ as

$$ E_1 : x \mapsto \begin{cases} 0 \rightarrow 0.2 \\ 1 \rightarrow 0.2 \end{cases} \quad E_2 : x \mapsto \begin{cases} 2 \rightarrow 0.4 \\ 3 \rightarrow 0.2 \end{cases} $$

We have $W(X) = 1$, $W(E_1) = 0.4$, and $W(E_2) = 0.6$. The entropy of the partition $\xi$ can be computed by:

$$ H(\xi) = H(0.4, 0.6) = 0.97 $$

This relation allows the transfer of certain probabilistic and information-theoretical notions to partitions of sets, where we can take advantage of the partial order between partitions.

**Definition 3.5** [Partial order on measurable partitions] The partial order relation
on $\Xi(X)$ is defined by $\xi \leq \xi'$ for $\xi, \xi' \in \Xi(X)$ if $H(\xi) \leq H(\xi')$, where $\Xi(X)$ denotes the set of partitions of $X$.

Specifically, it is straightforward to see that if every block of $\xi'$ is included in a block of $\xi$ then $\xi \leq \xi'$. The largest element of this lattice is the partition $\alpha_X = \{\{x\}| x \in X\}$; the least one is the partition $\omega_X = \{X\}$. The partial order relation on $(\Xi(X), \leq)$ is actually a bounded lattice. Consider a different partition from the previous example: $\xi' = \{E'_1, E'_2, E'_3\}$, where

$$
E'_1 : x \mapsto (0 \rightarrow 0.2), \quad E'_2 : x \mapsto (1 \rightarrow 0.2), \quad E'_3 : x \mapsto \begin{pmatrix} 2 \rightarrow 0.4 \\ 3 \rightarrow 0.2 \end{pmatrix}
$$

we have $W(E'_1) = 0.2$, $W(E'_2) = 0.2$, and $W(E'_3) = 0.6$, the entropy of the partition $\xi$ can therefore be computed by:

$$H(\xi') = H(0.2, 0.2, 0.6) = 1.37$$

According to the Definition 3.5, it is clear that for $\xi, \xi' \in \Xi(X)$, we have $\xi < \xi'$ since $H(\xi) < H(\xi')$. It is easy to see that the largest partition is the original concrete space $X$: $H(X) = H(0.2, 0.2, 0.4, 0.2) = 1.92$, and the least partition is the whole space itself (one block): $H(1) = 0$.

### 3.4 Abstract space

For our abstract domain we consider interval-based partitions of the measure space. Specifically, we describe the set of possible values in each partition $E_i$ as intervals, written as $[E_i](0 < i \leq n)$. Each interval-based partition still has a weight of itself denoted as $\mu_i(0 < i \leq n)$. A formal definition is presented as follows.

**Definition 3.6** [Abstract domain $X^\sharp$] An element of the abstract domain $x^\sharp_i \in X^\sharp$ is defined as a pair $(\mu_i, [E_i])$, where $\mu_i$ is the weight on the element, and $[E_i]$ is the interval-based partition obtained via the following steps:

- **Adjust the concrete space:** Consider the concrete space $X$ over the vector of variables denoted as $X = \langle x_1, \ldots, x_k \rangle$ (where $k$ denotes the number of the variables).

  Before making partition on the space, we sort the vector values of each element of the space based on the order of the variable in the vector and possible values of each variable: we put the high security variables in the left hand of the vector and the low ones in the right hand, i.e. for all $1 \leq i < k$, the security level of $x_i$ is greater than or equal to the security level of $x_{i+1}$; and then order all possible values of each variable as ascending, i.e. for any variable $x_i$, and for all $1 \leq j < m$: $v_{i,j} \leq v_{i,j+1}$, where $1 \leq i \leq k$, $m$ denotes the number of possible
values of variable $x_i$ in the concrete space.

- Make the partition: Consider $\xi = \{E_i|1 \leq i \leq n\}$ is a measurable partition of the \textit{adjusted} concrete space $X$. Such a \textit{sorted} partition is used to produce a \textit{disjoint} interval-based partition in the next step. Each element of $E_i$ is described as a vector of possible value sets due to the vector of variables $\vec{X} = \langle x_1, \ldots, x_k \rangle$ and a weight $\mu_i$ on it, \textit{i.e.}

$$E_i : \langle \{v^i_{11}, \ldots, v^i_{1m_1}\}, \{v^i_{21}, \ldots, v^i_{2m_2}\}, \ldots, \{v^i_{k1}, \ldots, v^i_{km_k}\} \rangle \rightarrow \mu_i$$

where for variable $x_j(1 \leq j \leq m), \{v^i_{j1}, \ldots, v^i_{jm_j}\}$ denote all the possible values of $x_j$ in $E_i$.

- Lift to interval-based partition: We lift each element $E_i$ of the partition to an interval-based partition $[E_i]$:

$$[E_i] : \langle I^i_1, I^i_2, \ldots, I^i_k \rangle \rightarrow \mu_i$$

where $I^i_j : [v^i_{j1}, v^i_{jm_j}] \ (1 \leq j \leq k)$ denotes the interval of possible values of variable $x_j$ within block $E_i$.

A component of the abstract domain $X^2$ is defined as a pair $\{([E_i], [E_i])|1 \leq i \leq n\}$, written as $\{([E_i] \rightarrow \mu_i|1 \leq i \leq n\}$ , where $\mu_i$ denotes the weight on $[E_i]$, $[E_i]$ is a partition on concrete space $X$ and is described by a set of intervals on possible values of each variable $x_j$ in the vector of program variables $\vec{X}$: \exists intervals $\langle I^i_j = [a^i_j, b^i_j]|1 \leq j \leq m, x_j \text{ in } \vec{X} \rangle$ on partition $E_i$, where $a^i_j = \inf\{v^i_{j1}, \ldots, v^i_{jm_j}\}$, $b^i_j = \sup\{v^i_{j1}, \ldots, v^i_{jm_j}\}$, and $\{v^i_{j1}, \ldots, v^i_{jm_j}\}$ denotes the possible values of $x_j$ inside the partition $E_i$, and $m_j$ is the number of all possible values of $x_j$ within $E_i$.

Intuitively, a single partition is constructed by choosing an interval on possible values for each variable. Note that we sort the possible values of each variable and the security levels in the variable vector before making partitions, and then lift each block of the partition into the interval describing one. Therefore, the elements of the interval-based partition are still disjoint.

Consider an example. Assume $x_1$ is a 3-bit variable in high security level, and $x_2$ is a low variable with initial value 0. Consider $\langle x_1, x_2 \rangle$ with joint distribution as follows:

$$\langle x_1, x_2 \rangle \mapsto \begin{pmatrix}
0, 0 & 0.2 & \langle 1, 0 \rangle & 0.2 \\
2, 0 & 0.4 & \langle 3, 0 \rangle & 0.2 \\
4, 0 & 0 & \langle 5, 0 \rangle & 0 \\
6, 0 & 0 & \langle 7, 0 \rangle & 0
\end{pmatrix}$$
and the partition is as
\[ E_1 : \langle x_1, x_2 \rangle \mapsto \langle \{0, 1\} x_1, \{0\} x_2 \rangle \rightarrow 0.4 \]
\[ E_2 : \langle x_1, x_2 \rangle \mapsto \langle \{2, 3\} x_1, \{0\} x_2 \rangle \rightarrow 0.6 \]
\[ E_3 : \langle x_1, x_2 \rangle \mapsto \langle \{4, 5, 6, 7\} x_1, \{0\} x_2 \rangle \rightarrow 0 \]

and the interval-based partition is thus
\[ [E_1] : \langle [0, 1], [0] \rangle \rightarrow 0.4 \]
\[ [E_2] : \langle [2, 3], [0] \rangle \rightarrow 0.6 \]
\[ [E_3] : \langle [4, 7], [0] \rangle \rightarrow 0 \]

where \( \mu_1 = 0.4, \mu_2 = 0.6, \mu_3 = 0 \). The partitions can be arbitrary, but human intervention can optimise the analysis. All strategies for partitioning on the space produce safe leakage bounds, but with different precision.

### 3.5 The Galois connection

The abstract space can be viewed as \( X(\alpha, \gamma) X^\sharp \), where \( X \) denotes the concrete space, \( X^\sharp \) denotes the abstract space, \( \alpha \) denotes the abstraction function, \( \gamma \) denotes the concretisation function as usual, and \( \langle \alpha, \gamma \rangle \) forms a Galois connection in which the abstraction function \( \alpha \) and concrete function \( \gamma \) are given as follows.

**Definition 3.7** [Abstraction function \( \alpha \)] The abstraction function \( \alpha \) is a mapping from concrete space \( X \) to the sets of interval-based partitions \( X^\sharp : X \rightarrow [X/\xi] \), where \( [X/\xi] = \{ (\mu_i, [E_i]) | 0 < i \leq n \} \), and \( [E_i] \) denotes an element of the interval-based partitions, and \( n \) is the number of the partitions. The rules for the abstraction function \( \alpha \) follow the steps to construct the abstract domain in Definition 3.6: adjust the vector values of each element of the concrete space, make the partition on the space, lift the elements of the partition into interval-based ones.

The measurability of this collection of partitions follows from their disjointness. However, non-injective semantics \( \llbracket \cdot \rrbracket \) can produce non-disjoint sets in some cases. For the case of non-disjoint partitions created by the semantic function \( \llbracket \cdot \rrbracket : X \rightarrow X \), we simply extend the space to eliminate the collisions. The method we consider is putting an index \( i \) on the overlapped part to distinguish them, where \( i \) is from the index of the partition \( E_i \) in which it is located.

**Example 3.8** Consider a simple program \( \llbracket \text{if } (x==0) x := x+1 \rrbracket \), the initial dis-
distribution of variable $x$ is as follows:

$$
x : \begin{pmatrix}
    0 & \rightarrow & 0.2 \\
    1 & \rightarrow & 0.2 \\
    2 & \rightarrow & 3 \rightarrow 0.2 \\
\end{pmatrix}
$$

consider the partition of the concrete space as:

$$E_1 : \langle \{0,0\} \rangle \rightarrow 0.2$$
$$E_2 : \langle \{1,2,3\} \rangle \rightarrow 0.8$$

The abstract space of $x$ is thus as

$$x^\sharp : \begin{pmatrix}
    [E_1] : \langle \{0,0\} \rangle \rightarrow 0.2 \\
    [E_2] : \langle \{1,3\} \rangle \rightarrow 0.8 \\
\end{pmatrix}$$

After executing the program (the detailed abstract semantic functions can be found in Section 3.6), the space is transformed to:

$$x^\prime \sharp : \begin{pmatrix}
    \langle 1,1 \rangle \rightarrow 0.2 \\
    \langle 1,3 \rangle \rightarrow 0.8 \\
\end{pmatrix}$$

The two elements of the final abstract space have a common point with value 1. However, we extend the space and view the common element with an index and being different. Therefore, the updated abstract space is still viewed as disjoint. It is clear that such treatment will not affect the safety of the leakage computation (see Proposition 4.6).

The concretisation of such abstract objects is thus the set of all measures matching the above conditions. We consider a sub-partition $\eta$ on each element of the final abstract space $X^\sharp$ to concretise it.

**Definition 3.9** [Concretisation function $\gamma$] The concretisation function $\gamma$ is a mapping: $X^\sharp \rightarrow \bigcup \{x | x \in [E_i] / \eta \}$, where the $[E_i]$ are the blocks of the abstract object $X^\sharp$, $\eta$ is a sub-partition on each block under uniform distribution. Specifically, the rules for the concretisation function $\gamma$ are presented as follows:

- Consider the abstract space as $X^\sharp = \{[E_i] \rightarrow \mu_i | 1 \leq i \leq n \}$ where $\forall 1 \leq i \leq n$, $[E_i]$ denotes the interval-based partition elements, $\mu_i$ is the weight of it, and $n$ is the number of the partition elements. Assume the vector of variables as $\vec{X} = \{x_1, x_2, \ldots, x_m\}$, $m$ is the number of variables, and the interval of possible values of $x_j$ ($1 \leq j \leq m$) within $[E_i]$ is $I_j = [a^j_i, b^j_i]$, then $[E_i]$ can be denoted as:

$$[E_i] = \langle [a^1_i, b^1_i], [a^2_i, b^2_i], \ldots, [a^m_i, b^m_i] \rangle$$
For each \( [E_i] \), we concretise each interval \([a_j, b_j]_{1 \leq j \leq m}\) due to each variable \(x_j\) into a set of values as:

\[
\hat{E}_i = \langle \{v_{i1}^j | a_{j1}^i \leq v_{i1}^j \leq b_{j1}^i \}, \ldots, \{v_{im}^j | a_{jm}^i \leq v_{im}^j \leq b_{jm}^i \} \rangle
\]

and \(\hat{E}_i \rightarrow \mu_i\).

Next we make a sub-partition \(\eta\) on each block. Consider the Cartesian product of each value set \(\{v_{ij}^1 | a_{ij}^1 \leq v_{ij}^1 \leq b_{ij}^1\}\) for all \(1 \leq j \leq m\) due to each variable \(x_j\):

\[
\hat{E}_i' = V_{i1}^1 \times V_{i2}^1 \times \ldots \times V_{im}^1
\]

where \(V_{ij}^j = \{v_{ij}^j | a_{ij}^j \leq v_{ij}^j \leq b_{ij}^j\}\), \(1 \leq j \leq m\). Let \(N_i = \prod_{j=1}^m |b_j - a_j + 1|\), i.e. \(N_i\) is the size of \(\hat{E}_i'\), and let \(e_{ki}^i\) denote the element of \(\hat{E}_i'\), i.e.

\[
\hat{E}_i' = \{e_{ki}^i | 1 \leq i \leq n, 1 \leq k_i \leq N_i, e_{ki}^i \in V_{i1}^1 \times \ldots \times V_{im}^1\}
\]

we make uniform distribution on \(\hat{E}_i'\); each element \(e_{ki}^i\) is thus with probability \(\frac{\mu_i}{N_i}\). We have:

\[
\hat{X} = \{e_{ki}^i \rightarrow \frac{\mu_i}{N_i} | 1 \leq i \leq n, 1 \leq k_i \leq N_i\}
\]

Specifically, for our purpose of leakage computation, we can just concentrate on the low output variable when making sub-partition \(\eta\) in the last two steps above. We extract the public output variable of interest from \(\hat{E}_i\) for our purpose of leakage computation. Assume \(x_j\) is the low output variable, then we write:

\[
\hat{E}_i(x_j) : \langle \{v_{ij}^j | a_{ij}^j \leq v_{ij}^j \leq b_{ij}^j\} \rangle \rightarrow \mu_i
\]

Let \(N_j^i = b_j^i - a_j^i + 1\) denote the size of the set \(\{v_{ij}^j | a_{ij}^j \leq v_{ij}^j \leq b_{ij}^j\}\) in \(\hat{E}_i\). We make a uniform distribution on it and get the normalised distribution \(x_j\) within \(\hat{E}_i\) as

\[
\left(\begin{array}{c}
v_{j1}^i \rightarrow \mu_i / N_j^i \\
v_{j2}^i \rightarrow \mu_i / N_j^i \\
\vdots \\
v_{jN_i}^i \rightarrow \mu_i / N_j^i
\end{array}\right)
\]

Therefore the distribution of \(x_j\) in the concretised space is:

\[
\hat{x}_j \mapsto \{v_{jki}^i \rightarrow \mu_i / N_j^i | 1 \leq i \leq n, 1 \leq k_i \leq N_j^i\}
\]

Let us go back to the previous example and do concretisation. We make a sub-partition on each abstract element after the execution of the program as:
\[
\gamma(x') = \gamma \left( \begin{aligned}
&\langle [1, 1] \rangle \rightarrow 0.2 \\
&\langle [1, 3] \rangle \rightarrow 0.8
\end{aligned} \right) = \left( \begin{aligned}
&1 \rightarrow 0.2 \\
&- - - - - - \\
&1 \rightarrow \frac{0.8}{3} \\
&2 \rightarrow \frac{0.8}{3} \\
&3 \rightarrow \frac{0.8}{3}
\end{aligned} \right)
\]

We also have the entropy computation as:
\[
\mathcal{H}(\gamma(x')) = 0.2 \log_2 0.2 + 0.8 \times \left( \frac{0.8}{3} \log_2 \frac{3}{0.8} \right) \times 3 = 1.99
\]
while \(x' \mapsto \left( \begin{aligned}
&1 \rightarrow 0.4 \\
&2 \rightarrow 0.4 \\
&3 \rightarrow 0.2
\end{aligned} \right)\), and
\[
\mathcal{H}(x') = 0.4 \log_2 \frac{1}{0.4} + 2 \times 0.2 \times \log_2 \frac{1}{0.2} = 1.52
\]
Therefore, \(\mathcal{H}(\gamma(x')) \geq \mathcal{H}(x')\), which also illustrates that our treatment of disjoint cases will not underestimate the entropy computation. The proof for the safety of the abstraction can be deduced from Proposition 3.10 and 4.6.

**Proposition 3.10** \(X\langle \alpha, \gamma \rangle X^\sharp\) is a Galois connection.

**Proof.** For complete lattices \(X\) and \(X^\sharp\), to prove the pair \(\langle \alpha, \gamma \rangle\) form a Galois connection, we need to prove that for all \(x \in X\) and \(x^\sharp \in X^\sharp\), \(x \sqsubseteq_X \gamma \circ \alpha(x)\) and \(\alpha \circ \gamma(x^\sharp) \sqsubseteq_X x^\sharp\).

The concrete space \(X\) with measures \((q_1, \ldots, q_m)\) can be viewed as a composition of two partitions denoted by: \(\xi \eta_0\), where \(\xi = \{E_1, \ldots, E_n\}\) with measures \(\mu_1, \ldots, \mu_n\) is the same partition as the abstraction \(X^\sharp\) given by \(\alpha\). Sub-partition \(\eta_0\) is the partition on \(\xi\) which recovers the abstract space \(X^\sharp\) to be the original concrete space, i.e. assume \(N_k\) is the number of the elements of \(E_k\) (where \(1 \leq k \leq n\) and therefore \(m = \sum_{k=1}^{n} N_k\) ), according to the definition of entropy of partitions (see Definition 3.4), for all \(x \in X\), we have

1. \(\mathcal{H}(x) = \mathcal{H}(\xi \eta_0) = \mathcal{H}(\xi) + \mathcal{H}(\eta_0|\xi)\)
2. \(= \mathcal{H}(\mu_1, \ldots, \mu_n) + \sum_{i=1}^{n} \mu_i \mathcal{H}\left(\frac{p_1}{\mu_1}, \ldots, \frac{p_n N_k}{\mu_n}\right)\)

where \(\frac{p_{11}}{\mu_1} = q_1, \ldots, \frac{p_{m N_k}}{\mu_n} = q_m\) are the measures over \(X\), and \(m = \sum_{k=1}^{n} N_k\).

The abstract space \(X^\sharp\) can be viewed as a partition \(\xi\) over \(X\) given by abstraction function \(\alpha\). The concretisation on \(X^\sharp\) is given by sub-partition \(\eta\) (uniformisation) on each block. Therefore \(\gamma \circ \alpha\) can be considered as the composition of partitions \(\xi \eta\), i.e. for all \(x^\sharp \in X^\sharp\), we have
(3) \( \mathcal{H}(\gamma(\alpha(x^2))) = \mathcal{H}(\xi\eta) = \mathcal{H}(\xi) + \mathcal{H}(\eta|\xi) \)

\[
\mathcal{H}(\frac{1}{N_i}, \ldots, \frac{1}{N_i}) = \sum_{i=1}^{n} \mu_i \mathcal{H}(\frac{1}{N_i}, \ldots, \frac{1}{N_i})
\]

since a uniform distribution maximises the entropy, we have

(5) \( H(\frac{1}{N_i}, \ldots, \frac{1}{N_i}) \geq \mathcal{H}(\frac{p_1}{\mu_i}, \ldots, \frac{p_{n_i}}{\mu_i}) \) for all \( 1 \leq i \leq n \)

By (1)–(5) it is easy to get that \( \mathcal{H}(x) \leq \mathcal{H}(\gamma(\alpha(x))), \) and similarly

\( \mathcal{H}(\alpha(\gamma(x^2))) \leq \mathcal{H}(x^2) \)

Therefore, \( x \sqsubseteq X \gamma \circ \alpha(x) \) and \( \alpha \circ \gamma(x^2) \sqsubseteq X^2 x^2. \)

### 3.6 Abstract Transformation

We have presented an abstract domain to approximate the concrete measure spaces. We need abstract semantic functions to produce the transformations of such abstract spaces. The leakage upper bound \( U_v \) on variable \( v \) will be discussed in Section 4. We just concentrate on the abstract space transformation here. Assume \([\cdot]\) is the concrete semantic function, and \([\cdot]^\sharp\) is the abstract semantic function, which drives the abstract spaces to transform.

In what follows, we present the abstract semantic operations.

#### 3.6.1 Arithmetic Expressions

In the concrete analysis, we denote the probability function of variable \( v \) as \( \mu_v \). Let \( \mu_v(c) \) be the probability that the value of \( v \) is \( c \), the domain of \( \mu_v \) will be the set of all possible values that \( v \) could be. The computation function \([e]\) of an arithmetic expression \( e \) defines the measure space that the arithmetic expression can evaluate to in a given set of environments.

Now consider the abstract arithmetic operation \( e^\flat(x^\sharp, y^\sharp) = z^\sharp \). We define the abstract arithmetic operation on each element \( \langle \mu_i, \{[a_v, b_v]_{v \in X}\} \rangle \rightarrow \langle \mu'_i, \{[a'_v, b'_v]_{v \in X}\} \rangle \) as:

\[
\begin{align*}
\mu'_i &= \mu_i \\
\text{if } v = z &\quad a'_v = \inf \{e(a_x, a_y), e(a_x, b_y), e(b_x, a_y), e(b_x, b_y)\} \\
\text{otherwise } &\quad a'_v = a_v, \quad b'_v = b_v
\end{align*}
\]

The arithmetic expression returns the probable values on the abstract lattice with the current context: the weight of each abstract element is unchanged, i.e., \( \mu'_i = \mu_i \);
the interval of variable \( v \) is updated by the intervals on the expression \( e \) if \( v \) suppose to be the assigned variable.

**Example 3.11** Consider expression \( [x + 2y] \), assume the initial distribution for \( (x, y) \) is \( \mu_{(x,y)} \) (the distribution function on a set \( M \) is viewed as a mapping: \( M \rightarrow [0,1] \)):

\[
\begin{align*}
\mu_{(x,y)} : & \{ (0,0) \rightarrow 0.1, (1,1) \rightarrow 0.1, (2,1) \rightarrow 0.1 \} \rightarrow E_1 \\
& \{ (3,2) \rightarrow 0.2, (4,2) \rightarrow 0.2, (5,2) \rightarrow 0.1 \} \\
& \{ (6,3) \rightarrow 0.1, (7,3) \rightarrow 0.1 \} \rightarrow E_2
\end{align*}
\]

- We take the partition:
  \[
  \begin{align*}
  & E_1([0,2])_x, [0,1])_y; \mu_1 = 0.3 \\
  & E_2([3,7])_x, [2,3])_y; \mu_2 = 0.7
  \end{align*}
\]

- Applying the arithmetic expression \( \mu_{[x+2y]} \), we get:
  \[
  \begin{align*}
  & E_1([0,4])_x, [x+2y])_y; \mu_1 = 0.3 \\
  & E_2([7,13])_x, [x+2y])_y; \mu_2 = 0.7
  \end{align*}
\]

### 3.6.2 Boolean Expressions

In the concrete probabilistic semantics, the transformation function for BExp returns the part of the space which matches the Boolean test \( b \). The abstract transformation function for Boolean expressions splits each abstract element and returns the part of each which makes the Boolean test \( b \) true. Specifically, given any abstract element \( (\mu_i, [E_i])_{1 \leq i \leq n} \) (where \( n \) is the number of partition elements), let \( \{a^j_i, b^j_i\}|1 \leq j \leq m\} \) denote the set of intervals for each variable \( x_j \) in \( X \) within \([E_i]\), we have:

\[
[b]_T^a(\{([E_i], \mu_i)|1 \leq i \leq n}\} = [b]_T^a(\{([a^j_i, b^j_i]|1 \leq j \leq m), \mu_i)|1 \leq i \leq n}\}
\]

where for each \( i \in [1, n] \), for all \( 1 \leq j \leq m \), \( a^j_i \leq v^j_i \leq b^j_i \), the Boolean test \( b \) evaluates to be true. Similarly, for test \( \neg b \), we have:

\[
[-b]_T^a(\{([E_i], \mu_i)|1 \leq i \leq n}\} = [-b]_T^a(\{([a^j_i, b^j_i]|1 \leq j \leq m), \mu_i)|1 \leq i \leq n}\}
\]

where for each \( i \), \( \mu^F_i = \mu_i - \mu^T_i \), and for all \( a^j_i \leq v^j_i \leq b^j_i \), the boolean test \( b \) evaluates to be false (i.e. \( \neg b \) to be true). Note that,

\[
f^2_{[-b]}(\{([E_i], \mu_i)) \cup f^2_{[\neg b]}(\{([E_i], \mu_i)) = \{([a^j_i, b^j_i]|1 \leq j \leq m), \mu^F_i\}
\]

18
where $1 \leq i \leq n$ and for all variable $x_j (1 \leq j \leq m)$ in $\vec{X}$ on each abstract element $[E_i]$

- $a_{ij}' = \min(a_{ij}^T, a_{ij}^F)$, $b_{ij}' = \max(b_{ij}^T, b_{ij}^F)$
- $\mu' = \mu_i^T + \mu_i^F = \mu$

3.6.3 Assignment

Assignment operation $[x := e]^{\sharp}$ updates the state such that the abstract domain of an assigned variable $x$ is mapped to the domain of an expression $e$. The weight of the partitions remain unchanged: $\mu'_i = \mu_i$ $(1 \leq i \leq n)$, and the interval of assigned variable $x$ is updated by the interval of the expression $e$ in the current environment: for all $1 \leq i \leq n$, $a_{x_i} = a_c^e$, $b_{x_i} = b_c^e$, where $[a_c^e, b_c^e]$ denotes the interval of expression $e$ in $[E_i]$.

3.6.4 Sequential Command

The sequential operator can be obtained by composition:

$$[c_1; c_2]^{\sharp}(X^{\sharp}) = [c_2]^{\sharp} \circ [c_1]^{\sharp}(X^{\sharp})$$

Assume $X' = [c_1](X^{\sharp})$, and $X'' = [c_2](X'')$, then $X''$ is the new state space by executing the sequential command over $X^{\sharp}$: $X^{\sharp} \xrightarrow{[c_1; c_2]^{\sharp}} X''$.

Example 3.12 Consider program $[x := x+y; y := x-y]$ as an example. Assume the initial distribution of $\langle x, y \rangle$ as

$$\mu_{x,y} \rightarrow \begin{pmatrix} \langle 2, 0 \rangle & \rightarrow & 0.2 & \langle 2, 2 \rangle & \rightarrow & 0.2 \\ \langle 2, 1 \rangle & \rightarrow & 0.3 & \langle 2, 3 \rangle & \rightarrow & 0.3 \end{pmatrix}$$

We take the partition as

- $E_1(\langle 2, 2 \rangle_x, [0, 1]_y) \rightarrow 0.5$
- $E_2(\langle 2, 2 \rangle_x, [2, 3]_y) \rightarrow 0.5$

After executing $c_1 : [x := x + y]$, we have

- $E_1(\langle 2, 3 \rangle_x, [0, 1]_y) \rightarrow 0.5$
- $E_2(\langle 2, 3 \rangle_x, [2, 3]_y) \rightarrow 0.5$

After executing $c_2 : [y := x - y]$, we have

- $E_1(\langle 2, 3 \rangle_x, [2, 2]_y) \rightarrow 0.5$
- $E_2(\langle 4, 5 \rangle_x, [2, 2]_y) \rightarrow 0.5$
i.e. the program copies the value of $x$ to $y$.

### 3.6.5 Conditional Statement

The Boolean tests split the abstract space into two parts. The weight on each part is the probability of the part of the space that makes the Boolean test be true or false. The if statement returns the sum of the two parts by executing the statements under true and false branches, therefore the intervals on variables are updated but the weight on each abstract block remains unchanged. Let $^+$ be an abstraction of the sum operation on measures, the abstract semantic function for if statements is given by:

$$[[\text{if } b \text{ then } c_1 \text{ else } c_2]]^+(X) = [[c_1]]^+ \circ [[b]]^+(X) + [[c_2]]^+ \circ [[\neg b]]^+(X)$$

We define the abstract operation $(X^t, X^T, X^F) \rightarrow X'$, where $X^T$ denotes the abstract object under true branch:

$$[[c_1]]^+ \circ [[b]]^+(X^t) = X^T_i\{([[a_i^T, b_i^T]|v \text{ in } \vec{X}), \mu_i^T]|1 \leq i \leq n\}$$

and $X^F$ denotes the abstract object under false branch:

$$[[c_2]]^+ \circ [[\neg b]]^+(X^t) = X^F_i\{([[a_i^F, b_i^F]|v \text{ in } \vec{X}), \mu_i^F]|1 \leq i \leq n\}$$

We have, $\forall v \in \vec{X}$:

$$(X^t, X^T, X^F) \rightarrow X'\{([[a_i^u, b_i^u]|v \text{ in } \vec{X}), \mu_i^u]|1 \leq i \leq n\}$$

for all $v \in \vec{X}, i \in N$, we take $^+$ as:

- $\mu_i^u = \mu_i^T + \mu_i^F = \mu_i$;
- $a_i^u = \min(a_i^T, a_i^F)$, and $b_i^u = \max(b_i^T, b_i^F)$ for all $v$ in $\vec{X}$ on each interval-based partition $[E_i]$.

#### Example 3.13

Consider program

```plaintext
if(x==0) then y:=0 else y:=1;
```

Assume $x$ is a 3-bit high security variable with distribution

$$
\begin{pmatrix}
0 & 0.1 & 1 & 0.1 & 2 & 0.1 & 3 & 0.1 \\
4 & 0.2 & 5 & 0.2 & 6 & 0.1 & 7 & 0.1
\end{pmatrix}
$$

$y$ is a 3-bit low security variable under any distribution. Let us take partition as:

$$
\begin{align*}
E_1([0,3], [0,7]) : \mu_1 &= 0.4 \\
E_2([4,7], [0,7]) : \mu_2 &= 0.6
\end{align*}
$$
It is clear that
\[
[y := 0]^T \circ [x := 0]^T (X^T) = X^{T^2} \begin{cases}
E^T_1 ([0, 0], [0, 0]) \to \mu_1^T \\
E^T_2 \phi \to \mu_2^T
\end{cases}
\]
and
\[
[y := 1]^T \circ [x \neq 0]^T (X^T) = X^{F^T} \begin{cases}
E^F_1 ([1, 3], [1, 1]) \to 0.4 - \mu_1^T \\
E^F_2 ([4, 7], [1, 1]) \to 0.6 - \mu_2^T
\end{cases}
\]
After applying the abstract operation $+^T$ on $X^{T^2}$ and $X^{F^T}$, we have:
\[
\begin{cases}
E'_1 ([0, 3], [0, 1]) : \mu_1' = 0.4 \\
E'_2 ([4, 7], [1, 1]) : \mu_2' = 0.6
\end{cases}
\]

3.6.6 Loops
The concrete semantics of loops are given as infinite sums of all the partitions which have found their way out in a recursive way. In order to consider the approximation of the fixpoint of some monotone operators on the concrete lattices and study the abstract operations, we rewrite the semantic function for loops as
\[
\text{[while } b \text{ do } c](X) = \neg b\lim_{n \to \infty} X_n
\]
following [28], where $X_n$ is defined recursively:
\[
X_0 = \lambda X.0, \quad X_{n+1} = X_n + [c] \circ [b](X_n)
\]
Consider $X^z$ as the abstraction of the measure space $X$, and let $+^z$ be an abstraction of the sum operation on measures, $\bot^z$ be an abstraction of the null measure, $X^z$ is considered recursively as:
\[
X_0^z = \lambda X.\bot, \quad X_{n+1}^z = X_n^z +^z [c] \circ [b]^z(X_n^z)
\]
We shall deal with it by using fixpoints in the abstract lattice $(X^z, \sqsubseteq)$. We repeatedly apply the upper bound step until a fixed point is reached:
\[
\text{[while } b \text{ do } c]^z(X^z) \triangleq [-b]^z(\text{lfp}^z \chi^z \mapsto X^z \sqsubseteq [c]^z([b]^z(\chi^z)))
\]
Assume
\[
\text{lfp}^z : (X^z \text{ monotone } X^z) \to X^z
\]
is an approximation of the least fixpoint, i.e. if $[\cdot]^z : X^z \to X^z$ is monotonic then we have the approximation relation
\[
[\cdot]^z(\text{lfp}^z(f)) \sqsubseteq \text{lfp}^z(f)
\]
Consider the abstract element \( ([a^v_i, b^v_i]|v in \bar{X}), \mu_i) \) so far, the resulting element of a loop iteration \( ([a^v_i, b^v_i]|v in \bar{X}), \mu_i) \), the abstract operation of while loop:

\[
((|a^v_i, b^v_i]|v in \bar{X}), \mu_i), \ ((|a^v_i, b^v_i]|v in \bar{X}), \mu_i') \rightarrow (|a^v_i, b^v_i]|v in \bar{X}), \mu_i'\)

is considered as the least upper bound:

\[
(|a^v_i, b^v_i]|v in \bar{X}), \mu_i) \sqcup (|a^v_i, b^v_i]|v in \bar{X}), \mu_i')
\]

where \( \forall i \in N \), we take \( \sqcup \) as:

- \( \mu_i'' = \mu_i = \mu_i' \)
- \( a^v_i = \min(a^v_i, a^{v'}_i), \text{ and } b^{v''}_i = \max(b^v_i, b^{v''}_i), \) for all \( v \in \bar{X} \) on \( |E_i| \)

Furthermore, in order to get a fast convergence via widening operators \([11]\), we need to consider the feasibility of a fixed point computation for the abstract semantics. To figure out the approximation of least fixpoint operation \( 1\text{fp}^\mathbb{F} \), we need an operation to be applied using widening operators \( \nabla \). By widening, the abstract function \( (X^\mathbb{F}, X^{\mathbb{F'}}) \rightarrow X^{\mathbb{F''}} \), i.e.

\[
((|a^v_i, b^v_i]|v in \bar{X}), \mu_i), \ ((|a^v_i, b^v_i]|v in \bar{X}), \mu_i') \rightarrow (|a^v_i, b^v_i]|v in \bar{X}), \mu_i'')
\]

can be noted as \( ((|a^v_i, b^v_i]|v in \bar{X}), \mu_i) \nabla (|a^{v'}_i, b^{v''}_i]|v in \bar{X}), \mu_i'') \) as standard. By induction, the upward iteration sequence is defined as \( X^\mathbb{F}_{n+1} = X^\mathbb{F}_n \nabla X^\mathbb{F}_{n+1} \) as usual, which is required to be eventually stationary for every ascending of chain elements and limited by a sound upper approximation of \( 1\text{fp}^\mathbb{F} \) for any sequence \( (X^\mathbb{F}_n)_{n \in N} \). Since our abstract domain is based on partitions of the measure space with coefficients of intervals of variable values, i.e. interval-based approximation, we choose the ordinary widening operator for intervals on the integers \( \nabla_I \) \([11]\): \( x \nabla_I y = \infty \), if \( x < y \), otherwise \( x \nabla_I y = y \), where \( x \) and \( y \) are two abstract element. The intuition behind applying this widening operator is that if a sequence of coefficients \( I_i \) keeps ascending, an upper approximation of it is obtained by using \(+\infty\). For all \( i \in N \), we take:

- \( \mu_i'' = \mu_i = \mu_i' \)
- Intervals \( I_i = [a_v, b_v]_{v \in \bar{X}} \) on \( |E_i| \) are given as standard:
  \[
  \nabla_I [a_v, b_v] = [a_v, b_v] \\
  [a_v, b_v] \nabla_I [a_v, b_v] = [a_v, b_v] \\
  [a_v, b_v] \nabla_I [a'_v, b'_v] = \text{if } a'_v < a_v \text{ then } -\infty \text{ else } a_v \\
  \text{if } b_v < b'_v \text{ then } +\infty \text{ else } b_v
  \]

By applying the widening operation for intervals \( \nabla_I \), we enforce the convergence and induce termination for intervals on each abstract block.
3.7 Soundness of Abstract Functions

In program analysis, abstraction describes a property-preserving contraction of a program’s model into one that is smaller and more suitable for automated analysis [8,9,10]. An abstraction must be sound, which means that properties proved true of the abstracted model must hold true in the original program model. Ideally, the abstract model should be most precise, or complete, with regards to the properties of interest.

First let us review some definitions and lemmas [8,9,34] to look at what a sound approximation on semantic operations means.

Concrete operations

In addition to building a model \(c \in C\) by \(\alpha(c) \in A\), let us consider modelling concrete computation steps, \(f : C \rightarrow C\). For instance, we can extend the concrete function to powersets (and thus the accumulating semantics) in the following standard way: for all subsets \(S\) in \(\mathcal{P}(C)\),

\[
\begin{align*}
    f^* : \mathcal{P}(C) &\rightarrow \mathcal{P}(C) \\
    f^*(S) &\equiv \{f(s) \mid s \in S\}
\end{align*}
\]

Abstract operations

It is then natural to define the abstract computation steps, \(f^* : A \rightarrow A\), and we want soundness as well.

Example 3.14 Consider a concrete domain \(N\) and the concrete operation increment \(\text{inc}: N \rightarrow N\) defined as \(\text{inc}(n) = n+1\). Then consider abstract domain Parity: \{any, none, even, odd\}, abstract operation of increment \(\text{inc}^* : \text{Parity} \rightarrow \text{Parity}\) is defined as:

\[
\begin{align*}
    \text{inc}^*(\text{even}) &\equiv \text{odd}, \quad \text{inc}^*(\text{odd}) \equiv \text{even} \\
    \text{inc}^*(\text{any}) &\equiv \text{any}, \quad \text{inc}^*(\text{none}) \equiv \text{none}
\end{align*}
\]

Definition 3.15 [Sound approximation c.f. [34]] Formally, for a Galois connection, \(\mathcal{P}(C)(\alpha, \gamma)A\), and functions \(f^* : \mathcal{P}(C) \rightarrow \mathcal{P}(C)\), \(f^* : A \rightarrow A\), \(f^*\) is a sound approximation of \(f^*\) iff

\[
(\alpha \circ f^*)(S) \subseteq A (f^* \circ \alpha)(S), \text{ for all } S \in \mathcal{P}(C)
\]

iff

\[
(f^* \circ \gamma)(a) \subseteq C (\gamma \circ f^*)(a), \text{ for all } a \in A
\]
This slightly abstract presentation exposes that \( \alpha \) is a “semi-homomorphism” [34] with respect to \( f^* \) and \( f^\sharp \), diagrammatically, see Figure 1.

\[
\begin{array}{c}
P(C) \\
\alpha \\
\downarrow f^* \\
P(C)
\end{array}
\begin{array}{c}
A \\
f^\sharp \\
\alpha
\end{array}
\]

Fig. 1. “Semi-homomorphism” of Abstraction Function

Furthermore, given the Galois connection, \( C(\alpha, \gamma)A \), and operation \( f^* : P(C) \rightarrow P(C) \), the most precise \( f^{\sharp}_{\text{best}} : A \rightarrow A \), which is sound with respect to \( f^* \) is

\[
f^{\sharp}_{\text{best}} = \alpha \circ f^* \circ \gamma
\]

**Lemma 3.16 (c.f. [34])** \( f^\sharp \) is sound with respect to \( f^* \) iff

\[
f^{\sharp}_{\text{best}}(a) = \alpha \circ f^* \circ \gamma(a) \sqsubseteq f^\sharp(a), \text{ for all } a \in A
\]

According to the discussion above, for a Galois connection, \( X(\alpha, \gamma)X^\sharp \), and semantic functions \([\cdot] : X \rightarrow X\), \([\cdot]^\sharp : X^\sharp \rightarrow X^\sharp\), \([\cdot]^\sharp \) is a sound approximation of \([\cdot] \) iff

\[
(\alpha \circ [\cdot]^\sharp)(S) \sqsubseteq (X^\sharp \circ \alpha)(S), \text{ for all } S \in X
\]

iff

\[
([\cdot] \circ \gamma)(A) \sqsubseteq (\gamma \circ [\cdot]^\sharp)(A), \text{ for all } A \in X^\sharp
\]

Furthermore, given the Galois connection, \( X(\alpha, \gamma)X^\sharp \), and operation \([\cdot] : X \rightarrow X\), the most precise \( [\cdot]^\sharp_{\text{best}} : X^\sharp \rightarrow X^\sharp\), which is sound with respect to \([\cdot]\) is

\[
[\cdot]^\sharp_{\text{best}} = \alpha \circ [\cdot] \circ \gamma
\]

**Lemma 3.17 (Soundness of composed functions)** Assume function \( f_1^\sharp \) is a sound abstraction of function \( f_1 \), and \( f_2^\sharp \) is a sound abstraction of \( f_2 \), then the abstract composed function \( f_1^\sharp \circ f_2^\sharp \) is sound with respect to the concrete composed function \( f_1 \circ f_2 \).

\[
\alpha \circ f_1 \sqsubseteq f_1^\sharp \circ \alpha \land \alpha \circ f_2 \sqsubseteq f_2^\sharp \circ \alpha \implies \alpha \circ (f_2 \circ f_1) \sqsubseteq (f_2^\sharp \circ f_1^\sharp) \circ \alpha
\]

**Proof.**

\[
f_1^\sharp \circ f_2^\sharp \circ \alpha \sqsupseteq (\alpha \circ f_1 \circ \gamma) \circ (\alpha \circ f_2 \circ \gamma) \circ \alpha \quad (\text{because } \alpha \circ f \circ \gamma \sqsubseteq f^\sharp)
\]

\[
= \alpha \circ f_1 \circ (\gamma \circ \alpha) \circ f_2 \circ (\gamma \circ \alpha) \sqsubseteq \alpha \circ f_1 \circ f_2 \quad (\text{because } \gamma \circ \alpha \sqsubseteq \text{Id}_c)
\]

\(\square\)
Theorem 3.18 shows the soundness of the abstraction \( \cdot \mid X^\sharp \): \( X^\sharp \to X^\sharp \) defined in previous sections of this chapter.

**Theorem 3.18 (Soundness of our abstract semantic functions) \( \cdot \mid X^\sharp \) is sound with respect to the concrete semantic functions \( \cdot \).**

**Proof.** To show the soundness of the abstraction is equivalent to prove that \( \cdot \mid X^\sharp \) is sound with respect to \( \cdot \) iff
\[
\cdot \mid X^\sharp \mid \text{best}(A) = \alpha \circ [\cdot] \circ \gamma(A) \subseteq [\cdot] \mid X^\sharp(A), \text{ for all } A \in X^\sharp
\]

**Assignment:** \( [x := e]^\sharp \) is sound with respect to \( [x := e] \) iff
\[
[x := e]^\sharp \mid \text{best} = \alpha \circ [x := e] \circ \gamma \subseteq_{X^\sharp \to X^\sharp} [x := e]^\sharp
\]
Assume the abstract domain of the expression \( e \) is denoted as
\[
M_e^\sharp = e^\sharp(X^\sharp) = \{(\mu_i, ([a_i^j, b_i^j]))|1 \leq i \leq n\}
\]
we have,
\[
\gamma(M_e^\sharp) = \bigcup_{1 \leq i \leq n} \{\mu_i/N_i, s_i^e | a_i^e \leq s_i^e \leq b_i^e, N_i = b_i^e - a_i^e + 1\}
\]
where \( n \) denotes the number of the abstract blocks, then we have:
\[
[x := e]^\sharp \mid \text{best}(x, M_e^\sharp)(X^\sharp)
\]
\[
= \alpha \circ [x := e] \circ [x := e] \circ \gamma(M_e^\sharp)(X^\sharp)
\]
\[
= \alpha([x := e][x, \bigcup_{i \in N} \{\mu_i/N_i, s_i^e | a_i^e \leq s_i^e \leq b_i^e\}]) (X^\sharp)
\]
\[
= \bigcup_{1 \leq i \leq n} \{\mu_i, E_i(x \mapsto [a_i^e, b_i^e])\}
\]
\[
\subseteq_{X^\sharp} [x := e]^\sharp(x, M_e^\sharp)(X^\sharp) \quad \text{(by the definition of } [x := e]^\sharp \text{ Section 3.6.3)}
\]

**Compositional operator** The soundness proof of the sequential abstract function is straightforward by Lemma 3.17.

**Conditional** To prove soundness of the abstract function for the if statement, we need to show that
\[
[\text{if } b \text{ then } c_1 \text{ else } c_2]^\sharp(X^\sharp) \subseteq [\text{if } b \text{ then } c_1 \text{ else } c_2]^\sharp(X^\sharp)
\]
According to the definition of \( [\cdot]^\sharp \mid \text{best} \), and definition of \( [b]^\sharp \) discussed in 3.6.2, and Lemma 3.16, we write:
\[
[b]^\sharp \mid \text{best} = \alpha \circ [b] \circ \gamma(X^\sharp)
\]
\[
= \alpha \circ [b] \circ \{\{\mu_i \circ N_i, (s_j^i | 1 \leq j \leq m, a_j^i \leq s_j^i \leq b_j^i)|1 \leq i \leq n\}
\]
\[
= \alpha \circ \{\{\mu_i \circ N_i, (s_j^T | 1 \leq j \leq m, a_j^T \leq s_j^T \leq b_j^T)|1 \leq i \leq n\}
\]
\[
= \{\{\mu_i, (a_i^T, b_i^T)|1 \leq j \leq m)\}|1 \leq i \leq n\}
\]
\[
25
\]
\[ [b] \subseteq [b]^2(X^\sharp) \quad (by \ the \ definition \ of \ [b] \ given \ in \ Section \ 3.6.2) \]

where \( m \) is the size of variable vector \( X \), \( n \) is the number of the abstract blocks.

Similarly, we have

\[ [-b]_{\text{best}} \subseteq [b]^2(X^\sharp) \]

By Lemma 3.17 and IH, we get that,

\[ [[c_1]_{\text{best}} \circ [b]_{\text{best}} + [[c_2]_{\text{best}} \subseteq [-b]_{\text{best}} \subseteq [c_1]_{\text{best}} \circ [b]_{\text{best}} + [c_2]_{\text{best}} \circ [-b]_{\text{best}} \]

(since \( +^\sharp \) is taken as the upper bound on the two branches, see 3.6.5). Therefore,

\[ [[\text{if } b \text{ then } c_1 \text{ else } c_2]_{\text{best}} \subseteq [[b]_{\text{best}} \]

- **Loop** According to Definition 3.15, to prove the soundness of \([\text{while } b \text{ do } c] \)

with respect to \([\text{while } b \text{ do } c], \) we need to show

\[ \alpha \circ [\text{while } b \text{ do } c] \subseteq [\text{while } b \text{ do } c] \circ \alpha \]

i.e. for all \( S \in X \)

\[ \alpha([\text{while } b \text{ do } c](S)) \subseteq [\text{while } b \text{ do } c] \circ \alpha(S) \]

Let \( F = \lambda \chi.X \cup [c](\llbracket b \rrbracket \chi) \), and \( F^\sharp = \lambda \chi.X^\sharp \cup [c] \llbracket [b] \rrbracket^\sharp \chi \). First we need to show \( F \) and \( F^\sharp \) are chain continuous and thus

\[ \text{lfp } F = \lim_{n \to \infty} (\lambda \chi.X \cup [c](\llbracket b \rrbracket \chi))^n = \bigsqcup_{n \geq 0} F_n(\bot) \]

\[ \text{lfp } F^\sharp = \lim_{n \to \infty} (\lambda \chi.X^\sharp \cup [c] \llbracket [b] \rrbracket^\sharp \chi))^n = \bigsqcup_{n \geq 0} F_n^\sharp(\bot) \]

Starting with \( \bot \), we apply the function \( F \) over and over again to build up a sequence of partial functions \( F_0, F_1, \ldots, F_n \). Let \( X \) be a poset, assume \( F : X \to X \), and

\[ F_0(\bot) \triangleq \bot \quad F_{n+1}(\bot) \triangleq F(F_n(\bot)) \]

Note that since \( \forall x \in X, \bot \subseteq x \), one has \( F_0(\bot) = \bot \subseteq F_1(\bot) \). By monotonicity of \( F \),

\[ F_n(\bot) \subseteq F_{n+1}(\bot) \Rightarrow F_{n+1}(\bot) = F(F_n(\bot)) \subseteq F_{n+2}(\bot) \]

Therefore, by induction on \( n \in N, \forall n \in N, F_n(\bot) \subseteq F_{n+1}(\bot) \). In other words, the elements \( F_n(\bot) \) do form a chain in \( X \).

By IH, we have:

(i) The base case is obvious when the function \( F^\sharp \) is applied 0 time:

\[ \alpha \circ \emptyset \subseteq \bot \circ \alpha \Rightarrow \alpha \circ (\lambda \chi.X \cup [c](\llbracket b \rrbracket \chi))^0 \subseteq (\lambda \chi.X^\sharp \cup [c] \llbracket b \rrbracket^\sharp \chi))^0 \circ \alpha \]

(ii) IH: Assume \( \forall n < k, \alpha \circ F_n \subseteq F_{n+1} \circ \alpha \)

then, \( \alpha \circ F_{k-1} \subseteq F_k \circ \alpha \) by IH \((k-1 < k)\)

and \( \alpha \circ F_1 \subseteq F_2 \circ \alpha \) by IH \((1 < k)\)
by Lemma 3.17 and definition of $F_n$ and $F^\sharp_n$, we have

$$\alpha \circ (F_{k-1} \circ F_1) \sqsubseteq (F^\sharp_{k-1} \circ F^\sharp_1) \circ \alpha \quad \text{iff} \quad \alpha \circ F_k \sqsubseteq F^\sharp_k \circ \alpha$$

Therefore, soundness of abstract function $F^\sharp$ is obtained as:

$$\alpha \circ \lambda \chi. X \cup \llbracket c \rrbracket \llbracket b \rrbracket \chi \n+1 \sqsubseteq \尖括号 \llbracket c \rrbracket \llbracket b \rrbracket \llbracket \llbracket c \rrbracket \llbracket b \rrbracket \chi \n+1 \circ \alpha$$

By Lemma 3.17, we have

$$\alpha \circ \llbracket \neg b \rrbracket \llbracket \llbracket c \rrbracket \llbracket b \rrbracket \chi \n \sqsubseteq \尖括号 \llbracket \neg b \rrbracket \llbracket \llbracket c \rrbracket \llbracket b \rrbracket \llbracket \llbracket c \rrbracket \llbracket b \rrbracket \chi \n \circ \alpha$$

i.e. $\alpha(\llbracket \text{while } b \text{ c}(S) \rrbracket) \subseteq \llbracket \text{while } b \text{ c}(\alpha(S)) \rrbracket$. This completes the proof of the soundness of the abstract semantic function for loops.

4 Entropy of Measurable Partitions and the Leakage Computation

Section 3.1 and Section 3.6 have described an abstraction on the measure space and the abstract semantic operations. In order to measure the information flow, we also need to consider an approximation on the leakage computation. We first consider the entropy of a partition (see Definition 3.4).

Consider a space $X$ as: $X \mapsto \begin{pmatrix} 0 \rightarrow 0.1 \\ 1 \rightarrow 0.6 \\ 2 \rightarrow 0.1 \\ 3 \rightarrow 0.2 \end{pmatrix}$, and a partition $\xi$ due to the parity of the possible values of $X$ as $X/\xi$:

$$E_1 : \{0,1\} \rightarrow 0.7, \ E_2 : \{2,3\} \rightarrow 0.3$$

i.e. $\mu(E_1) = 0.7, \ \mu(E_2) = 0.3$. The entropy of partition $\xi$ can be computed by:

$$\mathcal{H}(\xi) = \mathcal{H}(0.7, 0.3) = 0.88$$

We have defined that if $\xi, \xi' \in \Xi(X)$ and $\mathcal{H}(\xi) \leq \mathcal{H}(\xi')$, then $\xi \leq \xi'$. However, the partition entropy may underestimate the leakage computation. To get a conservative leakage analysis, we need a re-approximation on the final abstract space aiming to provide a safe (upper) bound on the leakage computation at the end. The basic method here is that we do a subpartition $\eta$ on each element of the final abstract objects obtained by the abstract operations. The subpartition we consider is by the way of performing a uniformisation over each element of the final partition $\xi' = [\llbracket c \rrbracket \xi$ according to the fact that uniform distribution maximises entropy. We call
a block uniform if all its states inside are with the same probability, i.e. all possible values of the distribution within that block are uniform.

**Definition 4.1** [Uniformalisation] Uniformalisation is defined as a transformation from each block of space of a variable to a space with uniform distribution on each block, i.e. each partition is sub-partitioned under a uniform distribution due to the values of a (low) variable of interest.

Let us look back to the previous example, after doing uniformalisation on the abstract space, we have:

\[ X/\xi \mapsto \begin{pmatrix} E_1\{0, 1\} \to 0.7 \\ E_2\{2, 3\} \to 0.3 \end{pmatrix} \xrightarrow{\text{Uniformalisation}} X' \mapsto \begin{pmatrix} 0 \to 0.35 \\ 1 \to 0.35 \\ 2 \to 0.15 \\ 3 \to 0.15 \end{pmatrix} \]

and the entropy of the updated space is

\[ H(X') = H(0.35, 0.35, 0.15, 0.15) = 1.88 \]

Using the definition of uniformalisation on measurable partitions, next let us consider the leakage upper bound \( U_v \) due to our abstract domain. Consider a subpartition \( \eta \) under a uniform distribution on the final partition \( \xi' \) in which any element \( E_i' \in \xi' \) comprises of \( N_i \) sub-elements with equal probability (where \( \|\|\xi = \xi' \), and \( \xi \) is the initial partition at the begin of the program), and let us attach to this partition the finite discrete uniform probability distribution whose corresponding entropy will be \( \log_2(N_i) \), where \( N_i \) denotes the size of the final abstract element \( E_i' \) due to the low security variable considered. We consider the entropy of the composition of partitions \( \xi' \) and \( \eta \) denoted by \( H(\xi'|\eta) \), which is defined as the sum of the well-defined entropy \( H(\xi') \) and the mean conditional entropy of \( \eta \) with respect to \( \xi' \): \( H(\eta|\xi') \) [32], i.e.

\[ H(\xi'|\eta) = H(\xi') + H(\eta|\xi') \]

where the conditional entropy is defined as follows.

**Definition 4.2** [Conditional entropy of partitions c.f. [32]] If \( \xi' \) and \( \eta \) are measurable partitions of a space \( X \), the conditional entropy \( H(\eta|\xi') \) is a non-negative measurable function on the factor space \( X/\xi \), called the conditional entropy of \( \xi' \) with respect to \( \eta \), which is defined as:

\[ H(\eta|\xi') = \sum_{i=1}^{n} \mu_i H(\eta) \]
where $\mu_i$ denotes the measure of each partition given by $\xi'$. The leakage upper bound is therefore defined as the entropy on the compositional partitions $\xi'\eta$. It is clear that the entropy of a measurable sub-partition $\eta$ on partition of $\xi'$ into $N_i$ sets is less than or equal to $\log_2(N_i)$ if every element of the sub-partition has measure $1/N_i$. This meets the intuition of our method: computation time is saved by doing abstract transformations on abstract partitions $\xi$ of the measure space, the safety of the leakage computation is guaranteed by considering the mean conditional entropy of a uniform partition $\eta$ with respect to the final abstract partition $\xi'$ returned by the execution of the program.

**Definition 4.3** The leakage upper bound $U_v$, for a given variable $v$, is given by the entropy of partition $\xi'\eta$, according to the definition of entropy on partitions and definitions of $\xi'$ and $\eta$, we have:

\[
U_v = H(\xi'\eta) = H(\xi') + H(\eta|\xi')
\]

\[
= H(\mu_1, \ldots, \mu_n) + \sum_{i=1}^{n} \mu_i H(\frac{\mu_i/N_i}{\mu_i}, \ldots, \frac{\mu_i/N_i}{\mu_i})
\]

\[
= H(\mu_1, \ldots, \mu_n) + \sum_{i=1}^{n} \mu_i H(\frac{1}{N_i}, \ldots, \frac{1}{N_i})
\]

\[
= H(\mu_1, \ldots, \mu_n) + \sum_{i=1}^{n} \mu_i \log_2(N_i)
\]

where $N_i$ is the size of the partition $E_i$, i.e. $N_i = (b_v - a_v + 1)$ for partition $E_i \in \xi'$.

**Example 4.4** Consider $[1 := h + 1]$ as a simple example to look at some intuition of the method. Assume the initial distribution of $h$ is

\[
\begin{pmatrix}
1 & 0.3 & 2 & 0.4 \\
3 & 0.1 & 4 & 0.1 \\
5 & 0.1
\end{pmatrix},
\]

and the distribution of $l$ is

\[
\begin{pmatrix}
0 & 0.5 \\
1 & 0.5
\end{pmatrix}.
\]

In order to show that the strategy of partitioning affects the precision of the leakage computation, we consider two different partitions...
\( \xi_1 \) and \( \xi_2 \) as follows.

- Consider the partitions \( \xi_1 \) on the joint distribution:

\[
\begin{align*}
E_1([1,1], [0,1]) &\rightarrow 0.3, \\
E_2([2,2], [0,1]) &\rightarrow 0.4, \\
E_3([3,5], [0,1]) &\rightarrow 0.3
\end{align*}
\]

The abstract semantic functions transform the above abstract objects into \( \xi'_1 \):

\[
\begin{align*}
E'_1([1,1], [2,2]) &\rightarrow 0.3, \\
E'_2([2,2], [3,3]) &\rightarrow 0.4, \\
E'_3([3,5], [4,6]) &\rightarrow 0.3
\end{align*}
\]

In the next step, we perform uniformisation on \( \xi'_1 \), and concentrate on the low security variable \( l \):

\[
\begin{align*}
X/\xi'_1 \xrightarrow{\text{uniformisation}} X/\xi'_1 \eta_1
\end{align*}
\]

Finally we do the leakage computation due to such spaces based on the leakage definition, the leakage upper bound is computed by:

\[
U_l = H(0.3, 0.4, 0.3) + 0.3 \times \log_2 3 = 2.0464
\]

- Next let us consider another strategy way of partitions \( \xi_2 \):

\[
\begin{align*}
E_1([1,3], [0,1]) &\rightarrow 0.8, \\
E_2([4,5], [0,1]) &\rightarrow 0.2
\end{align*}
\]

The abstract semantic functions transform the above abstract objects into \( \xi'_2 \):

\[
\begin{align*}
E_1([1,3], [2,4]) &\rightarrow 0.8, \\
E_2([4,5], [5,6]) &\rightarrow 0.2
\end{align*}
\]
After doing uniformisation, we have:

\[
\begin{array}{c}
X/\xi^2 \left( \begin{array}{c}
[2, 4]_l \rightarrow 0.8 \\
[5, 6]_l \rightarrow 0.2
\end{array} \right)
\end{array}
\overset{\text{Uniformisation}}{\Rightarrow}
\begin{array}{c}
X/\xi^2 \eta_2 \left( \begin{array}{c}
2 \rightarrow 0.8/3 \\
3 \rightarrow 0.8/3 \\
4 \rightarrow 0.8/3 \\
5 \rightarrow 0.2/2 \\
6 \rightarrow 0.2/2
\end{array} \right)
\end{array}
\]

and the leakage upper bound of variable \( l \) is given by:

\[
U_l = H(0.8, 0.2) + 0.8 \times \log_2 3 + 0.2 \times \log_2 2 = 2.1899
\]

- Furthermore, it is easy to get that the exact leakage due to this simple example is \( L = H(0.3, 0.4, 0.1, 0.1, 0.1) = 2.0464 \).

This simple example illustrates how uniformisation on abstract objects preserves the safety of leakage computations, and how the strategy for partitioning may affect the precision of the computation: the first strategy for partitioning in this example is much better than the second one since the first one is “closer” to the exact leakage. But both of them provide a safe leakage computation, i.e. don’t underestimate the leakage. The precision of the partitioning strategy depends on the initial distribution of high security inputs and also the programs, but not in general.

Next, in order to show how the abstract transformation functions and leakage computations work, let us look at a simple example with while loop.

**Example 4.5** Consider a loop program

\[
l := 0; \text{ while}(l < h) \text{ do } l++;
\]

assume \( h \) is a 3-bit high security variable with distribution

\[
\begin{array}{c}
0 \rightarrow 0.1, 1 \rightarrow 0.1 \\
2 \rightarrow 0.1, 3 \rightarrow 0.1 \\
4 \rightarrow 0.2, 5 \rightarrow 0.2 \\
6 \rightarrow 0.1, 7 \rightarrow 0.1
\end{array}
\]

is a 3-bit low security variable with an initial value of 0.

Let us take partition \( \xi \) as:

\[
\begin{cases}
E_1([0, 3]_h, [0, 0]_l) \rightarrow 0.4, \\
E_2([4, 7]_h, [0, 0]_l) \rightarrow 0.6
\end{cases}
\]

31
Using the abstract operations, we present the transformation due to each iteration as follows. Let $[\text{while}]_i^q$ denote the abstract operation of the loop due to the $i^{th}$ iteration, and $X_i^q$ denote the updated abstract space due to the $i^{th}$ iteration.

$$
[\text{while}]_0^q (X_i^q) = \lfloor (1 < h) \rfloor (X_0^0) + q \lfloor 1 + q \rfloor (\lfloor 1 < h \rfloor (X_0^0))
$$

$$
= X_1^q \{ \{[0,3]_h, [h,0]_l \} \rightarrow \mu^F, \langle [4,7]_h, [h,0]_l \} \rightarrow \mu^F_l \} + q

\{ \{[0,3]_h, [1, h]_l \} \rightarrow 0.4 - \mu^F, \langle [4,7]_h, [1,1]_l \} \rightarrow 0.6 - \mu^F_l \}

= X_2^q \{ \{[0,3]_h, [0,3]_l \} \rightarrow 0.4, \langle [4,7]_h, [1,1]_l \} \rightarrow 0.6 \}
$$

$$
[\text{while}]_1^q (X_i^q) = \lfloor (1 < h) \rfloor (X_1^q) + q \lfloor 1 + q \rfloor (\lfloor 1 < h \rfloor (X_1^0))
$$

$$
= X_2^q \{ \{[0,3]_h, [h,3]_l \} \rightarrow \mu^F, \langle [4,7]_h, [h,2]_l \} \rightarrow \mu^F_l \} + q

\{ \{[0,3]_h, [1, h]_l \} \rightarrow 0.4 - \mu^F, \langle [4,7]_h, [2,2]_l \} \rightarrow 0.6 - \mu^F_l \}

= X_3^q \{ \{[0,3]_h, [0,3]_l \} \rightarrow 0.4, \langle [4,7]_h, [2,2]_l \} \rightarrow 0.6 \}
$$

$$
[\text{while}]_2^q (X_i^q) = \lfloor (1 < h) \rfloor (X_2^q) + q \lfloor 1 + q \rfloor (\lfloor 1 < h \rfloor (X_2^0))
$$

$$
= X_3^q \{ \{[0,3]_h, [h,3]_l \} \rightarrow \mu^F, \langle [4,7]_h, [h,4]_l \} \rightarrow \mu^F_l \} + q

\{ \{[0,3]_h, [1, h]_l \} \rightarrow 0.4 - \mu^F, \langle [4,7]_h, [3,3]_l \} \rightarrow 0.6 \}

= X_4^q \{ \{[0,3]_h, [0,3]_l \} \rightarrow 0.4, \langle [4,7]_h, [3,3]_l \} \rightarrow 0.6 \}
$$

$$
[\text{while}]_3^q (X_i^q) = \lfloor (1 < h) \rfloor (X_3^q) + q \lfloor 1 + q \rfloor (\lfloor 1 < h \rfloor (X_3^0))
$$

$$
= X_4^q \{ \{[0,3]_h, [h,3]_l \} \rightarrow \mu^F, \langle [4,7]_h, [h,4]_l \} \rightarrow \mu^F_l \} + q

\{ \{[0,3]_h, [1, h]_l \} \rightarrow 0.4 - \mu^F, \langle [4,7]_h, [5,3]_l \} \rightarrow 0.6 - \mu^F_l \}

= X_5^q \{ \{[0,3]_h, [0,3]_l \} \rightarrow 0.4, \langle [4,7]_h, [4,3]_l \} \rightarrow 0.6 \}
$$

$$
[\text{while}]_4^q (X_i^q) = \lfloor (1 < h) \rfloor (X_4^q) + q \lfloor 1 + q \rfloor (\lfloor 1 < h \rfloor (X_4^0))
$$

$$
= X_5^q \{ \{[0,3]_h, [h,3]_l \} \rightarrow \mu^F, \langle [4,7]_h, [h,7]_l \} \rightarrow \mu^F_l \} + q

\{ \{[0,3]_h, [1, h]_l \} \rightarrow 0.4 - \mu^F, \langle [4,7]_h, [5,7]_l \} \rightarrow 0.6 - \mu^F_l \}

= X_6^q \{ \{[0,3]_h, [0,3]_l \} \rightarrow 0.4, \langle [4,7]_h, [4,7]_l \} \rightarrow 0.6 \}
$$

A fixpoint is reached at the end. Next, we concentrate on the low variable, do uniformisation on each block to concretise the final space, and have:

$$
\begin{align*}
\{ [0,3]_l \rightarrow 0.4, [4,7]_l \rightarrow 0.6 \} & \xrightarrow{\text{Uniformisation}} \mu_l \mapsto \begin{pmatrix}
0 \rightarrow 0.4/4 & 1 \rightarrow 0.4/4 \\
2 \rightarrow 0.4/4 & 3 \rightarrow 0.4/4 \\
4 \rightarrow 0.6/4 & 5 \rightarrow 0.6/4 \\
6 \rightarrow 0.6/4 & 7 \rightarrow 0.6/4
\end{pmatrix}
\end{align*}
$$

Finally we show the entropy of $l$ via the leakage definition after uniformisation.
i.e. the leakage upper bound is computed by

\[ U_l = \mathcal{H}(0.4, 0.6) + 0.4 \times \log_2 4 + 0.6 \times \log_2 4 = 2.97 \]

That is of course not as good as the exact computation:

\[ L = \mathcal{H}(0.1, 0.1, 0.1, 0.1, 0.2, 0.2, 0.1, 0.1) = 2.92 \]

but still offers a reasonable upper bound. This example also shows our abstraction techniques improve the time consuming problem of leakage analysis (4 iterations needed).

The measurement produced by our technique should be an upper bound on the actual information flow, so that our technique can overestimate the amount of information leaked, but can never underestimate it. To address this concern, Proposition 4.6 is presented to show formally the soundness of our approximation for abstract leakage analysis.

**Proposition 4.6** Consider the final abstract space obtained by the semantic functions \([\mathcal{H}]\) to be a finite measurable partition \(\xi\). After performing uniformalisation sub-partition \(\eta\) on these output abstract objects \(\xi\), we get safe leakage upper bound by computing the entropy of \(\xi \eta\).

**Proof.** Consider a distribution \(\mu\) over a set of events \(E\) containing \(N\) elements and let \(\xi = \{E_1, \ldots, E_n\}\) be the partitions of this set, \(\eta\) be a sub-partition on \(\xi\). Assume \(N_k\) is the number of elements of \(E_k\), we have \(N = \sum_{k=1}^{n} N_k\), and let \(\mu_k\) be the weight of the partition \(E_k\), \((k = 1, \ldots, n)\), and for all \(1 \leq k \leq n\), \(\{\mu_{kj}|1 \leq j \leq N_i\}\) denote the normalised probability distribution under \(E_k\). Let the elements of \(E\) be labeled from 1 to \(N\), i.e. \(E = \{e_1, \ldots, e_N\}\) and the elements of \(E_k\), \((k = 1, \ldots, n)\) be labeled from 1 to \(N_k\). According to the definition of entropy of partitions [32,31], the entropy of such space over \(E\) can be computed by:

\[ H(\xi \eta) = H(\xi) + H(\eta|\xi) \]

\[ = H(\mu_1, \ldots, \mu_n) + \sum_{i=1}^{n} \mu_i H\left(\frac{\mu_1}{\mu_i}, \ldots, \frac{\mu_i}{\mu_i}\right) \]

Now consider that \(\eta\) is under a uniform distribution over each element of partition \(\xi\), since the number of elements in partition \(E_k\) is \(N_k\), the entropy of the space after uniformalisation on the partitions can be computed by:

\[ H(\xi \eta) = H(\mu_1, \ldots, \mu_n) + \sum_{i=1}^{n} \mu_i H\left(\frac{1}{N_i}, \ldots, \frac{1}{N_i}\right) \]

Since a uniform distribution maximises entropy, it is clear that for each partition,

\[ \sum_{i=1}^{n} H\left(\frac{1}{N_i}, \ldots, \frac{1}{N_i}\right) \geq \sum_{i=1}^{n} H\left(\frac{\mu_1}{\mu_i}, \ldots, \frac{\mu_i}{\mu_i}\right) \]
Therefore we get $\bar{H}(\xi \eta) \geq H(\xi \eta)$, i.e. we show that uniformisation on the partitions keeps the entropy computation safe.

## 5 Related Work

Quantitative information flow has recently become an active research topic in the computer security community. The precursor for this work was that of Denning in the early 1980’s. Denning [13] presented that the data manipulated by a program can be typed with security levels, which naturally assumes the structure of a partially ordered set. Moreover, this partially ordered set is a lattice under certain conditions [12]. However, he did not suggest how to automate the analysis. In 1987, Millen [27] first built a formal correspondence between non-interference and mutual information, and established a connection between Shannon’s information theory and state-machine models of information flow in computer systems. Later related work is that of McLean and Gray and McIver and Morgan in 1990’s. McLean presented a very general Flow Model in [26], and also gave a probabilistic definition of security with respect to flows of a system based on this flow model. The main weakness of this model is that it is too strong to distinguish between statistical correlation of values and casual relationships between high and low object. It is also difficult to be applied to real systems. Gray presented a less general and more detailed elaboration of McLean’s flow model in [18], making an explicit connection with information theory through his definition of the channel capacity of flows between confidential and non-confidential variables. Webber [36] defined a property of n-limited security, which took flow-security and specifically prevented downgrading unauthorized information flows. Wittbold and Johnson [37] gave an analysis of certain combinatorial theories of computer security from information-theoretic perspective and introduced non-deducibility on strategies due to feedback. Gavin Lowe [22] measured information flow in CSP by counting refusals. Aldini and Di Pierro [1] introduced a method of quantifying information flow on a probabilistic process system, The method is based on process similarity relation with regard to an approximation of weak bisimulation of CCS. Backes [2] gave a definition for measuring the quantity of information flow within interactive settings by measuring the distance between different behaviours of high user from low user’s views. McIver and Morgan [25] devised a new information theoretic definition of information flow and channel capacity. They added demonic non-determinism as well as probabilistic choice to *while* program thus deriving a non-probabilistic characterization of the security property for a simple imperative language. There are some other attempts
in the 2000s: Di Pierro, Hankin and Wiklicky [14,15,16] gave a definition of probabilistic measures on flows in a probabilistic concurrent constraint system where the interference came via probabilistic operators. Clarkson et al. [7] suggested a probabilistic beliefs-based approach to non-interference. This work might not works for most of situations. Different attackers have different beliefs, therefore the worst case is required. Boreale [3] studied the quantitative models of information leakage in the process calculi. Clark, Hunt, and Malacaria [4,5,6] presented a more complete reasonable quantitative analysis for a particular program in imperative languages but the bounds for loops are over pessimistic. Malacaria [23] gave a more precise quantitative analysis of loop construct but it is hard to automate. McCamant and Ernst [24] investigated techniques for quantifying information flow revealed by complex programs by building flow graphs and considering the weight of the maximum flow over it. Köpf and Basin [19] developed a quantitative model for assessing a system’s vulnerability to adaptive side-channel attacks. Recently, Smith [35] considered a new foundation based on the concept of vulnerability, which measures uncertainty by applying Rényi’s min-entropy rather than Shannon entropy. Backes, Köpf, and Rybalchenko [?] presented an automatic method for information flow analysis that discovered what information was leaked by considering equivalence relations on program variables.

6 Conclusions and Future Works

We aimed to develop automatic and high quality leakage analysis tools for programs. In [29], we devised a method that implements Kozen’s [21] concrete probabilistic semantics, and applied this semantics to calculate leakage. In this paper, we present an approach to provide an approximation on such leakage analysis.

We have presented a very basic abstraction on the distribution transformations and leakage computations in this chapter. Intuitively, the information flow is introduced by data dependence information among variables with different security levels. One possible future direction for work due to this chapter is the possibility of a representation of the dependence flow graph to capture the properties of secure information flow within the programs as a further abstraction of the leakage analysis. It is possible to extend the previous work to graphs further by building the dependence flow graph which leads to a simple algorithm, based on abstraction interpretation, for providing more efficient information flow measurement analysis. Building this flow graph allows a tool to automatically compute the bound on information flow, achieving more abstract and efficient analysis compared to
the abstraction described in this chapter. The basic idea is to use the dependence flow graph as our intermediate representation. The dependence flow graph can be viewed as a data structure in which edges represent dependencies between operations. Dependence flow graphs integrate data and control dependence information into a single structure, making efficient algorithms for program analysis and optimization possible. For every dependence edge in the data dependence graph, there is a corresponding path in the dependence flow graph. Unlike data dependence graphs, dependence flow graphs are executable. A dependence flow graph consists of a set of nodes representing functional operators and a set of edges representing the dependencies and precedence relations that exist between those operations. If there is a dependence between two statements in the source program, then there must be a path between the corresponding nodes in the dependence flow graph. Control dependencies are represented by switches and merges that route data and resource dependencies to several destinations based on the control condition’s value. We need to develop a simple algorithm to build a flow graph and then compute the information flow in a graph efficiently. We consider the above discussion as one direction for future work for this paper.

**Acknowledgement**

We thank Geoffrey Smith, Ton Coolen, and the anonymous reviewers for useful comments. This research was partially supported by the EPSRC grant EP/C009967/1 Quantitative Information Flow.

**References**


[34] D. Schmidt. From trace sets to modal-transition systems by stepwise abstract interpretation, 2001.

